AD/A-005 826

A PHONOLOGICAL RULES SYSTEM

J. A. Barnett

System Development Corporation

Prepared for:

Advanced Research Projects Agency

24 January 1975

DISTRIBUTED BY:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD/A005 826

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A PHONOLOGICAL RULES SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Technical - 1974 6. PERFORMING ORG. REPORT NUMBER TM-5478/000/00
7. AUTHOR(s) Barnett, J. A.		8. CONTRACT OR GRANT NUMBER(s) DAHC15-73-C-0080
9. PERFORMING ORGANIZATION NAME AND ADDRESS System Development Corporation 2500 Colorado Avenue Santa Monica, California 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Code 5D30
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 24 January 1975 13. NUMBER OF PAGES 50
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Cleared for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

automatic speech processing
LISP extensions

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(53)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TM-5478/000/00

A PHONOLOGICAL RULES SYSTEM

J. A. BARNETT

24 JANUARY 1975

THIS REPORT WAS PRODUCED BY SDC IN PERFORMANCE OF CONTRACT NO. DAHC15-73-C-0080, ARPA ORDER NO. 2254, PROGRAM CODE NO. 5D30.

ia

System Development Corporation
2500 Colorado Avenue • Santa Monica, California 90406

TABLE OF CONTENTS

Appendices

1. INTRODUCTION

A phonological rules system has been implemented as a language extension of SDC INFIX LISP[1]. The system can be used in two modes: (1) as an interactive rule tester, and (2) as a library of functions with other LISP programs. The key language capabilities are:

- definitions of phonological rules,
- definitions of ordered rule application,
- definitions of unordered rule application,
- definitions of nondeterministic rule application,
- lexicon definitions,
- sub-lexicon definitions, and
- multiple forms for words in the lexicon.

The key system capabilities and features are:

- ability to edit and recompile all definitions,
- ability to output symbolics to a terminal, printer, or disk file in a format that allows recompilation,
- ability to selectively test a rule or group of rules against a single form, a lexicon entry, a sub-lexicon, or the entire lexicon, and
- all phonological rule definitions and all rule applying subr definitions are compiled rather than interpreted.

This document describes the phones and their features, the individual commands, the available editor, and the system. Appendix 1 gives a formal BNF definition of the commands.

2. PHONES AND THEIR FEATURES

Appendix 2 summarizes the available phonetic symbols and their features. The symbols used are from the ARPAbet[1]. Each phone has a KIND. The possible values of KIND are VOWEL, BOUNDARY, and CONST (consonant).

2.1 VOWEL FEATURES

All vowels have the feature VOICE and a stress level. There are three levels of stress: 0, 1, and 2. Stress level 0 means reduced, stress level 1 means unstressed, and stress

---

[1] The ARPAbet is a phonetic representation agreed upon by a group of ARPA contractors for transmitting phonetic strings to and from a computer.

level 2 means stressed. Schwa (AX) is always assumed to be reduced. Other vowels may have their level of stress specified by following the vowel name with a colon and one of the integers 0, 1, or 2. For example, a reduced IH is written IH:0, an unstressed OW is written OW:1, and a stressed ER is written ER:2.

2.2 BOUNDARY FEATURES

There are two boundary phones: *, which is a syllable boundary, and #, which is a word boundary. Neither is marked with any features other than BOUNDARY.

2.3 CONSONANT FEATURES

All other phones are consonants and therefore have the feature CONST. Each consonant also has a CLASS feature, may be marked VOICE or not, and may have a place of articulation specified. The possible values of CLASS are NASAL, PLOS, FRIC, AFRIC, GLIDE, LATERAL, CENTRAL, and MISC. The respective meanings are nasal, plosive, fricative, affricative, glide, L, R, and miscellaneous. The presence of the feature VOICE means that the phone is voiced. All the consonants except HH and the glottal stop (Q) have a place of articulation feature. The values of PLACE are LABIAL, DENTAL, ALVEOLAR, ALVPAL (alveolar-palatal), VELAR, and PALATAL.

2.4 PHONOLOGICAL SPELLINGS OF LEXICAL FORMS

Several system commands input an argument form spelled in ARPAbet. The format of the spelling is a sequence of phone names enclosed in parentheses. For example, a phonological spelling of HAVING is

    (HH AE:2 V*IH:0 NX)

and a spelling of BIG HOUSES is

    (B IH G#HH AW:2 Z * AX Z)

Several points should be noted about the input format. First, it is not necessary to enter the left and right word boundaries (#) because the system automatically adds them. Second, if a vowel other than schwa is entered without an explicit stress level (:0, :1, or :2 following the vowel) then the stress level is assumed to be 1. Third, all contiguous pairs of symbols in a phonological spelling must be separated from each other by one or more blanks unless at least one of them is "(", ")", "*", "#", or ":". In the latter case, though not required, blanks are permissible.

3. RULE DEFINITION

This section describes the command that causes a phonological rule to be compiled into the system. A formal syntax description of <rule> is given in Appendix 1. The rule language is also described by Barnett in [2].

A rule definition is introduced by $ followed by a blank and the rule name. The rule name is an identifier -- a sequence of letters, digits, and periods, the first of which is not a digit. The rest of the rule consists of a left side, an equal sign (=), a right side, and an optional conditional phrase. The right side of a rule specifies a pattern (schemata) of phone sequences. If the right side matches (properly describes) an input phonetic sequence, the transformation described by the left side is performed on the input sequence. The conditional, if present, describes additional criteria that must be met for the transformation to be made. For example, a simple rule for changing a T to a flap (DX) is

```
$ FLAP DX=VOWEL/T/VOWEL IF STRESS@1 GR STRESS@3;
```

The rule name is FLAP. The right side is VOWEL/T/VOWEL and describes a sequence of any vowel followed by T followed by any vowel. The left side is DX, and the associated transformation is to substitute DX for the sequence between the / pair, in this case for T. The conditional is IF STRESS@1 GR STRESS@3 and it means that the transformation should be done only if the stress level of the first phone (first vowel) in the matched sequence is greater than the stress level of the third phone (second vowel) in the matched sequence. For example, the input sequence

```
IH:2 T IY:1
```

would be transformed by FLAP to

```
IH:2 DX IY:1
```

However, the following sequences would not be transformed:

```
IH:2 R T IY:1
IH:1 T IY:2
```

The first sequence is not matched by the right side because of the presence of R. The second sequence is matched by the right side of FLAP but fails the conditional test because the stress level of the first vowel is not greater than the stress level of the second vowel.

The following subsections describe the right side, left side, and conditional parts of rules and present some examples.

3.1 RIGHT SIDE OF RULES

The right side of a rule describes phonetic sequences and is therefore a pattern. This pattern consists of three parts: a left context, a nucleus, and a right context. Any of the three parts may be vacuous. However, at least one of the three parts must not be vacuous. The nucleus part of the right side matches the portion of the input sequence that is affected by the transformation performed by this rule. The left and right contexts specify the necessary environment in which the nucleus is to occur. Normally, the nucleus is delimited by a / pair. If the pair is not present, then the whole right side is assumed to be the nucleus and the left and right contexts are assumed to be vacuous. The following paragraphs describe the constituents (<right-part>s) that make up the nucleus and the left and right contexts.

3.1.1 Phone Name

The name of a phone may be used to specify the occurrence of that phone in the input string. For example, M means the occurrence of M in the input string, and IY means the occurrence of IY in the input string (with any stress level). If it is desired to restrict the stress level of a vowel to a particular value, then follow the vowel name with a colon and an explicit stress level. Thus, to specify the occurrence of an IH with 2 stress, write IH:2.

3.1.2 Single Feature Specification

The occurrence of any phone with a specific feature may be specified. For instance, BOUNDARY may be used to specify the occurrence of either * or #. Similarly, CONST specifies the occurrence of any consonant, and VOWEL specifies the occurrence of any vowel. In a like manner, the values of CLASS and PLACE (such as NASAL, which specifies the occurrence of M, N, or NX, or VELAR, which specifies the occurrence of either NX, K, or G) may be used. Also, VOICE may be used to specify the occurrence of a voiced phone. To specify the occurrence of any vowel with a specific stress level, write VOWEL followed by a colon and an explicit stress level. For example, to specify the occurrence of a reduced vowel, write VOWEL:0. Any of the described specifications may be optionally preceded by + or -. Plus (+) merely emphasizes that the specification following is necessary; the + is therefore just window dressing. Minus (-), on the other hand, means that the specification following must not occur. Thus, -VOICE specifies the occurrence of any unvoiced phone, -VELAR specifies the occurrence of any phone except NX, K, or G, and -R specifies the occurrence of any phone except R.

3.1.3 Multiple Feature Specification

The occurrence of a phone that simultaneously possesses several features may be specified by a <feature-bundle>. A feature bundle is represented as a sequence of feature specifications (including phone names, choices and other feature bundles) enclosed in parentheses. The included specifications may be preceded by + or -. For example,

(FRIC LABIAL) and (FRIC+LABIAL)

both specify the occurrence of a labial fricative, i.e., either F or V. The feature bundles

(FRIC-LABIAL) and (+FRIC-LABIAL)

both specify the occurrence of a non-labial fricative, i.e., TH, S, SH, DH, Z, or ZH. An example of a nested feature bundle is

(FRIC-(LABIAL+VOICE))

This specifies the occurrence of any fricative that is not both voiced and labial. It could have been written more simply as

(FRIC -V)

3.1.4 Choice Specification

The occurrence of a phone in the input sequence may be specified as a <choice> among several specifications. The individual choice specifications may be phone names, a feature, or a feature bundle. The choices are separated by OR. For example, the choice

NASAL OR (PLOS+VOICE)

specifies the occurrence of a nasal or a voiced plosive. Thus, it would match any of N, M, NX, B, G, or D. For another example, the choice

GLIDE OR R OR L

specifies the occurrence of any of Y, W, R, or L. The equivalent of an AND operator is provided through the feature bundle mechanism.

3.1.5 Specification of Optional Occurrences

The optional occurrence of a phone in the input string can be specified by an <optional> phrase. The word OPT is followed by a phone name, feature, feature bundle, or choice. For example, the specification sequence

VOWEL, OPT R, T

would match both of the following input strings:

IY R T and IY T

If a phone is detected in the input string that matches the specification of an optional phrase, then it is passed over before a matching of the rest of the input string to the rest of the pattern is attempted. This means that there is no automatic backup. To illustrate this, the pattern sequence

VOWEL, OPT VOICE, R

would match the input string

OW AX R

but would not match the input string

OW R D

In fact, the above pattern sequence would match nothing that did not also match the pattern sequence

VOWEL, VOICE, R

3.1.6 Specification of Repeated Occurrences

The repeated occurrence of phones that match a particular set of criteria may be specified by a <repeat> phrase. The phrase is introduced by the word REP followed by the minimum acceptable number of occurrences (a non-negative integer) and the match criterion. The match criterion may be a phone name, feature, feature bundle, or choice. For example, a pattern that matches any monosyllabic word is

#, REP 0 CONST, VOWEL, REP 0 CONST, #

In the above, the input sequence is specified to include a beginning word boundary (#) followed by zero or more consonants, a vowel (at any stress level), zero or more trailing consonants, and an ending word boundary. In the next example, the pattern sequence will match the beginning consonant cluster and vowel in a syllable whose initial cluster contains at least two phones.

BOUNDARY, REP 2 CONST, VOWEL

Repeat phrases, like optional phrases, do no backup. The repeat matches as many phones in the input string as possible. (If at least the specified minimum number of occurrences are found, then matching of the rest of the input string to the rest of the right side continues.) For example, the pattern sequence

REP 0 CONST, R

would not match anything because R is a CONST, and as such would be passed over by the repeat. This may be remedied by rewriting the pattern sequence as

REP 0 (CONST-R), R

This second pattern sequence does the job because in English two Rs can not occur in the same consonant string unless separated by a syllable or word boundary.

3.1.7 Examples of Complete Right Side Patterns

The complete right side of a rule consists of a nucleus and a left and a right context. Each of these constituents of the right side comprises a sequence of phone names, features, feature bundles, choices, optional phrases, and repeat phrases. The nucleus is normally delimited by a / pair. The nucleus is the portion that will be replaced if the rule applies. The members of the sequence are separated from each other by commas. If a / separates two specifications, then a comma should not be used.

VOWEL/T/VOWEL
The left context is the one-element sequence, VOWEL. The nucleus is the one-element sequence, T. The right context is the one-element sequence, VOWEL. If a rule with this right side matches a pattern, then the nucleus (T) would be

replaced by the sequence generated by the rule's left side.

/D OR T, BOUNDARY/Y
In this example, the left context is vacuous, and the right context is the one-phone sequence, Y. The nucleus is the two-element sequence D OR T, BOUNDARY. The nucleus matches any one of these four input sequences:

D *, D #, T *, and T #

VOWEL, NASAL or /VOWEL, NASAL/
Both the left and right contexts are vacuous in these two equivalent pattern sequences. The nucleus is the two-element sequence VOWEL, NASAL. These two examples illustrates the point that if the / pair is omitted from the rule's right side, then the entire right side is the nucleus.

NASAL//PLOS
In this example, the nucleus is vacuous. The / pair merely marks a place at which the left side can insert a phone string if the rule matches.

3.1.8 Indices of Right Side Parts

Components of a conditional phrase and a rule's left side can reference features of the phones that were matched by the rule's right side. The referenced phone is specified by @ followed by a strictly positive integer. Each right part in the right side is assigned an index number starting from one. For example, in the pattern sequence

VOWEL,OPT BOUNDARY/IY OR IH,REP 1 LABIAL/(PLOS-VELAR),#

there are six right parts:

1 VOWEL,
2 OPT BOUNDARY,
3 IY OR IH,
4 REP 1 LABIAL,
5 (PLOS-VELAR), and
6 #

Though optional or repeat phrases are assigned index numbers, the features of the phones they match may not be references because it is indeterminate whether they matched anything at all and, if so, how many phones were matched. Example uses of indexed references and their meanings are:

NAME@1 name of phone matched by the first right part

KIND@2	kind of phone matched by the second right part
PLACE@3	place of phone matched by the third right part
CLASS@2	class of phone matched by the second right part
VOICE@1	voicing of phone matched by the first right part
-VOICE@3	inverse of the voicing of phone matched by the third right part
STRESS@1	stress level of phone matched by the first right part

3.2 CONDITIONAL PART OF RULES

Use of a <conditional> with a rule is optional. If the <conditional> is omitted, the only criterion for a rule's matching an input string is that the right side of the rule properly describe (match) the string. If a conditional phrase is used, it presents additional criteria that must also be satisfied for the rule to match the input string.

The form of a conditional is the word IF followed by the body of the conditional. The body is a series of relationships separated by the word AND or OR (inclusive). AND binds tighter than OR. Thus, if r1, r2, and r3 are relations, then the meaning of

    r1 OR r2 AND r3   is   r1 OR (r2 AND r3)

To overcome the normal binding scheme, parentheses may be used to explicitly group the relations and operators. For instance, to achieve the other interpretation of the above example, write

    (r1 OR r2) AND r3

Relations may either test a feature of a single phone or compare the features of two phones. Tests are usually indicated by using one of the operators EQ, NQ, GQ, LQ, GR, and LS. An example of a relation is

    PLACE@1 NQ PLACE@3

which is satisfied if the place of articulation of the phone matching the first right part is not equal to the place of articulation of the phone matching the third right part. (See section 3.1.8 for an elaboration on the meaning of indexed references to right parts such as @1 and @3 in this example.) Another example of a relation is

    CLASS@2 EQ FRIC

which is satisfied if the class of the phone that matched the second right part is FRIC. At first, this may seem unnecessary. Could not the second right part just have been written FRIC and the relation not used? To answer this question, consider this example right side and conditional:

CONST,PLOS OR FRIC IF VOICE@1 OR CLASS@2 EQ FRIC

Together, the right part and conditional match a two-phone sequence if either the first phone is voiced and the second phone is a plosive or fricative, or if the first phone is any consonant and the second phone is a fricative. To write such complex matching criteria as this, it is necessary to have conditionals and to be able to write tests against constants.

Relations that test a phone's kind, class, place, and name may be written in one of two ways as demonstrated by the above examples. In the first way, an indexed feature category is compared by the operator EQ or NQ to a constant value in that feature category. Examples are:

KIND@4 EQ VOWEL
CLASS@1 NQ PLOS
PLACE@2 EQ VELAR
NAME@3 NQ IY

The second method of comparison matches the feature values of two different phones. Examples are:

KIND@3 NQ KIND@1
CLASS@1 EQ CLASS@2
PLACE@3 NQ PLACE@2
NAME@2 EQ NAME@1

Relations involving stress level may be made in a similar manner. In addition, the operators GQ, LQ, GR, and LS may be used. Some examples are:

STRESS@2 GR 0
STRESS@1 LS STRESS@3

The possible constant values of stress level are 0, 1, and 2.

Since there are no symbols for constant values of voicing, the single-phone tests are written as in these examples:

VOICE@2
-VOICE@3

Comparisons between the voicing of two phones are written as in these examples:

```
VOICE@2 EQ VOICE@3
VOICE@2 NQ VOICE@1
```

Only the operators EQ and NQ may be used in voicing comparisons.

3.3 LEFT SIDE OF RULES

The <left-side> of a rule specifies the sequence of phones that is to replace the sequence of phones that was matched by the nucleus part of the rule's right side. If the sequence to be substituted is vacuous, then the left side is NIL. For example,

```
$ DEGEM NIL=/CONST/OPT BOUNDARY,CONST
            IF NAME@1 EQ NAME@3;
```

This is the version of the standard degemination rule that removes the first consonant of a doubled pair whether or not they are separated by a word or syllable boundary. DEGEM produces the following transformations:

```
T T  to  T
S*S  to  *S
M#M  to  #M
```

If the sequence to be substituted is not vacuous, then it is represented by a sequence of <left-part>s separated by commas. (In a prior version of the system, a rule could have multiple sequences of left parts. See [2].) The allowed kinds of left parts are consonant names, boundary names, vowel specifiers, and constructed consonants. The following paragraphs describe the different kinds of left parts and present some examples of complete rules.

3.3.1 Consonant and Boundary Name Left Parts

A consonant or boundary name may be used as a left part. For example, in the rule

```
$ FLAP DX=VOWEL/T OR D/VOWEL
          IF STRESS@1 GR STRESS@3
```

DX is a left part. It is substituted for an intervocalic T or D whenever the stress level of the first vowel exceeds that of the second vowel.

In addition, consonants and boundaries may be specified as left parts by use of index numbers. For example,

$ FOO 3,2=VOWEL/BOUNDARY,R OR L/VOWEL;

In this example, the two phones matching the nucleus are transposed. The index 3 references the phone R or L, and the index 2 references the phone * or # that matched BOUNDARY. Two transformations that would be produced by this rule are

AH*R IY to AH R*IY
AX#L UW to AX L#UW

Index references are restricted to phones that are matched by the right side. Thus, it is illegal to reference, say, the first phone following the string matched by the pattern.

3.3.2 Vowel Specification Left Parts

The specification of a vowel in the reconstruction sequence may be accomplished in a variety of ways. The vowel name and the stress level may be given explicitly, the name and/or stress level may be borrowed using indices, or the stress level may be borrowed implicitly. The various techniques are demonstrated by the following examples.

$ R1 IH:1=/VOWEL/OPT BOUNDARY,N;
In this example, an IH with stress level one is substituted for the vowel that matched the first right part.

$ R2 IH@1=/VOWEL/OPT BOUNDARY,N;
Like the above example, IH is substituted for the vowel that matched the first right part. However, the stress level is borrowed from the original vowel by @1. Thus, if the input string were IY:0*N then rule R1 produces IH:1*N and rule R2 produces IH:0*N.

$ R3 1,R=/VOWEL,*,ER:0/;
As with consonant and boundary names, vowel names may be referenced by an index. In this example, the left part 1 is whatever vowel (and its stress level) that is matched by the first right part. Thus, R3 would transform the input string AH:2*ER:0 to AH:2 R.

$ R4 1@3,R=VOWEL,*,ER;
This example is like R3 except that only the vowel name is borrowed from the phone matching the first right part. The stress level is borrowed from the ER that matches the third right part (by the @3). Thus, R4 would transform the input string AH:2*ER:0 to AH:0 R.

$ R5 IH=/IY OR EH/N;
In this example, only a vowel name (IH) is given as a left part. When this form of left part is used, the stress level is borrowed from the phone that matches the first vowel specification in the nucleus. These are two transformations that result from the application of rule R5:

```
IY:1 N  to  IH:1 N
EH:0 N  to  IH:0 N
```

Another example of implicit stress borrowing is rule R6:

```
$ R6 ER=VOWEL,R,*,VOWEL;
```

R6 transforms the input string EH:2 R*AX to ER:2 because the stress level is borrowed from the first vowel.

Two things should be noted in using vowel-specifying left parts: (1) it is illegal to write such things as AX@3 or AX:0 because AX is automatically given a stress of zero, and (2) when an implicit stress level is borrowed, it is never taken from a vowel that was matched by a repeat or optional phrase; it is borrowed from the first other right part in the nucleus that specifies a vowel (if there is a choice, then each choice must specify a vowel).

3.3.3 Constructed Consonants

Some consonant phones may be constructed by specifying their features. Specifically, the class, place of articulation, and voicing must be specified, in that order, and enclosed by parentheses. Some examples of constructed consonants are:

```
(NASAL ALVEOLAR VOICE)
(PLOS PLACE@2 -VOICE)
(CLASS@3 LABIAL VOICE@1)
```

The first example is equivalent to having written N. The class specification may be either a class name or the word CLASS followed by @ and an index. Examples from the above are NASAL, PLOS, and CLASS@3. In the last case, the class of the constructed phone is made the same as the class of the phone that matched the third right part. In a similar manner, the place of articulation may be either a place name or the word PLACE followed by @ and an index. Examples from the above are ALVEOLAR, PLACE@2, and LABIAL. Voicing of a constructed consonant is specified by either VOICE or -VOICE with the obvious meanings, or by use of a borrowed voicing, e.g., VOICE@3 or -VOICE@2. An example of a rule that uses a constructed consonant is:

```
$ JHA 2,(AFRIC ALVPAL VOICE@1)=T OR D,BOUNDARY,Y
```

This rule would make transformations such as

```
D*Y  to  *JH
T*Y  to  *CH
```

A caution should be observed when using constructed consonant forms -- namely, that there exists a phone with the specified class, place of articulation, and voicing. Because of this, it is illegal to construct a consonant in the class MISC. It is the user's responsibility to guard against the generation of illegal phones. The system does little run-time checking.

4. LEXICON AND SUB-LEXICON DEFINITION

This section describes the commands that are used to define lexicon entries and form sub-lexicons. Appendix 1 gives a formal syntax description of these forms (<lexicon> and <sub-lexicon>).

4.1 DEFINITION OF LEXICON ENTRIES

There are three basic forms of the lexicon command: (1) add, replace, or modify a lexicon entry; (2) augment a lexicon entry; and (3) print out a lexicon entry. All lexicon commands begin with the word LEX and end with a semicolon. A lexicon entry is identified by a word, e.g., an identifier such as HELLO or ONE.TWO.THREE. Associated with each word in the lexicon are one or more ARPAbet spellings. Each of these spellings is called a lexical base form or, more simply, just a form.

The basic command that adds a new word and its forms to the lexicon is the word LEX followed by the word and the forms spelled in ARPAbet (as described in Section 2.4). To enter a phonological spelling of the word TOTAL, input this command:

```
LEX TOTAL (T OW:2 T*AX L);
```

Recall that the exterior word boundaries are automatically added so that the actual spelling is

```
#T OW:2 T*AX L#
```

If it is desired to enter the word TOTAL with two forms, then the forms are separated by commas; for example:

```
LEX TOTAL (T OW:2*T AX L), (T OW:2 T*UH:0 L);
```

With either of the above examples, if TOTAL was already in the lexicon, all existing forms would be deleted, and the new definition would comprise the entire set of forms for this word.

Various commands allow the forms to be referenced individually. The language mechanism is the word followed by a colon and an index number. Thus, given the second definition above of TOTAL,

```
TOTAL:1 is #T OW:2 T*AX L#
```

and

```
TOTAL:2 is #T OW:2 T*UH:0 L#
```

It is also possible to selectively alter the definition of a particular form as opposed to redefining the whole entry. For example, after

```
LEX TOTAL:2 (T OW:2 T*AH:0 L);
```

TOTAL:1 is unaffected but

```
TOTAL:2 is now #T OW:2 T*AH:0 L#
```

When using this form of the lexicon command, the index must reference an existing form or be one greater than the number of forms currently in existence. In the latter case, the new form is added to the lexicon entry.

New forms may easily be added to the lexicon entry. Assume that the command

```
LEX CUP (K AH P);
```

has been executed, and then the command

```
LEX CUP+(K AH B), (K UH P);
```

is entered. There are now three forms of the word CUP:

```
CUP:1 is #K AH:1 P#
CUP:2 is #K AH:1 B#
CUP:3 is #K UH:1 P#
```

Thus, new forms are added by using + followed by one or more ARPAbet spellings.

The lexicon command is also used to output forms to the user's terminal. Given the above definition of CUP, the command

LEX CUP;

would output all three spellings. The command

LEX CUP:2;

would only output the spelling of CUP:2.

4.2 DEFINITION OF SUB-LEXICONS

A sub-lexicon is defined by a <sub-lexicon> command. The format of the command is the word SLEX followed by the sub-lexicon name (an identifier), and the constituents separated by commas. For example,

SLEX EXAMPLE TOTAL, CUP:2;

The sub-lexicon EXAMPLE is defined to contain all the forms of the word TOTAL but only the second form of the word CUP. A word or its forms may appear in any number of sub-lexicons

A sub-lexicon definition is maintained in symbolic form. Therefore, the actual forms that constitute the sub-lexicon are those in existence when the sub-lexicon is referenced -- not necessarily the same as those in existence when the sub-lexicon was defined.

5. RULE APPLYING SUBRS

When a rule is operated on a phonetic input string, it is usually desired to try it on all substrings, not just the input as a whole. Therefore, given the rule,

$ FLAP DX=VOWEL,OPT BOUNDARY/T OR D/OPT BOUNDARY,VOWEL;

the desired transformation of

#WH AH:1 T#AX#D EY:2# is #WH AH:1 DX#AX#DY EY:2#

Thus, it is necessary to define the algorithm by which a rule is tried on substrings of the input.

Also, it is usual to operate rules in groups. Such a group is called a rule set. At issue is the method of defining the constituents of a rule set and the order dependencies of the set members. The system provides three methods of specification: (1) ordered rule sets -- normally used with "obligatory" rules, (2) unordered rule sets -- normally used with "optional" rules, and (3) nondeterministic rule sets -- normally used for fun. All three types are defined by

<subr> commands.

The following subsections describe substring selection algorithms and the methods of defining rule-applying subrs.

5.1 SUBSTRING SELECTION

An input string is a phonological spelling of a word or a sequence of words. It is unusual that a rule will match (or was intended to match) the entire input string. Therefore, the rule set appliers must select substrings as possible candidates on which to try the rules. By example, the possible substrings of

```
#K AA:2 N#D UW:1#
```

are

```
#K AH:2 N#D UW:1#
K AH:2 N#D UW:1#
AH:2 N#D UW:1#
N#D UW:1#
#D UW:1#
UW:1#
#
```

Because neither optional nor repeat phrase (in the right side of rules) perform backtracking and because rules permit arbitrary parts of the input string to exist to the right of the substring matched by the right part, the above set of substrings is sufficient. With the different types of rule-applying subrs, the interaction (ordering) of members of the rule set with substrings of the input and substrings of the derived strings may differ as described below.

5.2 ORDERED RULE SUBRS

An ordered rule-applying subr is defined by the word SUBR followed by its name (an identifier) and a sequence of <subr-part>s separated by commas and terminated with a semicolon. The word ORDERED may follow the word SUBR but is not necessary -- ordered subrs are the default. In operation, the first subr part is applied to each substring of the input in turn, left to right. Then the second subr part is applied to each substring, etc. If a rule in a subr part matches the input string, it is immediately transformed by the rule, and the rest of the processing continues at the same phone position in the derived string. Thus, the subr parts and the rules they comprise are treated as obligatory transformations. The algorithm is presented symbolically in

```
DEFINE RUNRULE(SUBRPARTLIST,ARPASPELL)
    DO SPELL:=ARPASPELL;
       LSPELL:=LENGTH(SPELL);
       FOR SUBRPART IN SUBRPARTLIST
        DO FOR I:=1 STEP 1 UNTIL I>LSPELL
            DC SUBRPART(SPELL,I);
           END FOR;
       END FOR;
    END RUNRULE;
```

Figure 1
ORDERED RULE APPLICATION ALGORITHM

Figure 1. Application of a subr part SUBRPART(SPELL,I), operates on the substring starting at the Ith phone position. As a side effect, the values of the variables SPELL and LSPELL (spelling length) may be altered.

A subr part may be a rule name or a oneof, allof, if, or unless phrase. A rule name used as a subr part means simply operate the rule at the proper times. The following paragraphs describe the other kinds of forms that may be used as subr parts.

5.2.1 Oneof Subr Parts

A oneof subr part is introduced by the word ONEOF followed by a parenthesized list of one or more subr parts separated by commas. For example,

ONEOF(R1,R2)

The rule names R1 and R2 are the embedded subr parts. A oneof phrase runs its embedded subr parts (in the left to right order of their appearance) on the currently visible substring. If any rule in a subr part matches the input substring, then after the completion of the operation of that subr part, the rest of the oneof phrase in which it is embedded is skipped. Therefore, in the above example, if R1 matches the input substring, then R2 is not operated.

5.2.2 Allof Subr Parts

An allof subr part is introduced by the word ALLOF followed by a parenthesized list of one or more subr parts separated by commas. For example,

AILOF(ONEOF(R1,R2), R3, R4)

The embedded subr parts are the oneof phrase ONEOF(R1,R2) and the rule names R3 and R4. An allof phrase operates the embedded subr parts in the specified order, left to right. In the above example, R1 is operated; if it matches the input substring, then R3 and R4 are operated on the transformed string. Otherwise, rules R2, R3, and R4 are operated in that order. Recall that all embedded subr parts are run on the substring starting at the same phone position. Therefore, caution should should be exercised to ensure compatibility of operation with your original intentions.

5.2.3 If and Unless Phrases

If and unless subr parts provide for standard if-then and if-then-else control logic. An if phrase is introduced by the word IF followed by a subr part, the word THEN, another subr part, and optionally the word ELSE followed by yet another subr part. Unless phrases are identical in format to if phrases except that they are introduced by the word UNLESS instead of the word IF. If s1, s2, and s3 are subr parts, then the possible formats are:

```
IF s1 THEN s2
UNLESS s1 THEN s2
IF s1 THEN s2 ELSE s3
UNLESS s1 THEN s2 ELSE s3
```

For the first format, s1 is run. If any rule run in s1 matches the input, then s2 is run. Otherwise, s2 is skipped. For the second format, s1 is run. If no rule run in s1 matches the input, then s2 is run. Otherwise, s2 is skipped. For the third format, s1 is run. If any rule run in s1 matches the input, then s2 is run. Otherwise, s3 is run. For the fourth format, s1 is run. If any rule run in s1 matches the input, then s3 is run. Otherwise, s2 is run. All applicable embedded subr parts (s1 and s2 or s3) are run on the substrings at the same phone position. (If s1 matches the input, the input is transformed before the operation of s2 or s3.)

5.3 UNORDERED RULE SUBRS

An unordered rule-applying subr is defined by the word SUBR followed by the word UNORDERED, its name (an identifier), and the name of the rules in the rule set separated by commas. For example:

```
DEFINE RUNRULE(RULESET,ARPASPELL)
    DO LOOPER(RULESET,ARPASPELL,1);
    END RUNRULE;

DEFINE LOOPER(RULESET,ARPASPELL,INDEX)
    DO PRINT(ARPASPELL);
       FOR I:=INDEX STEP 1 UNTIL I>LENGTH(ARPASPELL)
        DO FOR RULE IN RULESET
            DO CHANGES:=RULE(ARPASPELL,I);
               IF CHANGES
                  THEN DO NEWSPELL:=MAKECHANGE(ARPASPELL,
                                                 CHANGES);
                       LOOPER(RULESET,NEWSPELL,I);
                     END;
            END FOR;
        END FOR;
    END LOOPER;
```

Figure 2
UNORDERED RULE APPLICATION ALGORITHM
Figure 2

SUBR UNORDERED XYZ R1,R2,R3;

XYZ is defined as an unordered rule-applying subr that operates the rule set that consists of the rules R1, R2, and R3. Figure 2 shows the rule-application algorithm. When a rule is run on an input string (CHANGE:=RULE(ARPASPELL, I)), it is passed two arguments: (1) the total input string and (2) the phone position at which the current substring starts. The value of the rule is false if the rule does not match the current input string. If the rule does match, then the value is the set of changes that should be produced by the rule's left side. The function, MAKECHANGE, takes two arguments: (1) the original spelling, and (2) the set of change instructions. The value is a new spelling with the changes made. The original spelling (the value of ARPASPELL) is not altered. As can readily be seen by tracing through LOOPER, each rule in RULESET is run at each position of the input string, with and without other applications of rules in the set.

The critical difference between an unordered and a nondeterministic rule subr is the following case. Assume that rule R1 applies to the input substring starting at phone position i1, and rule R2 would apply to the input substring as produced by the transformation done by the left side of R1, but starting at phone position i2 where i2<i1. Then an unordered subr will not operate R2 after R1, while a

nondeterministic subr will. The advantage of the unordered subr is that it is much faster. In almost no case will the difference in output be noticeable.

5.4 NONDETERMINISTIC RULE SUBRS

The format of a nondeterministic rule subr is identical to that of an unordered rule subr except that the keyword NODETERM is used rather than UNORDERED. For example:

```
SUBR NODETERM XYZ R1,R2,R3;
```

defines the rule-applying subr XYZ with rule set R1, R2, and R3. Figure 3 shows the algorithm used for nondeterministic rule application. In operation, a nondeterministic subr attempts to apply every member of the rule set against every possible substring of the input and the derived strings. This process continues until no new spellings can be generated and then terminates.

6. THE QUERY COMMAND

A ? may be used to query the system for the names of the defined entries. The four forms of the <query> command are:

```
? RULES;
? SUBRS;
? SLEXS;
? LEX;
```

The respective meanings are: (1) output the names of all defined rules, (2) output the names of all defined rule-applying subrs, (3) output the names of all defined sub-lexicons, and (4) output the names (not the spellings) of all defined lexicon entries. All output is to the user's terminal. For more detailed output, see Section 4 on the lexicon command and Section 8 on the output commands.

7. RUN COMMANDS

There are three commands that may be used to run a rule subr against a form, a word in the lexicon, a sub-lexicon, or the whole lexicon. The commands are <run>, <joe>, and <mary>.[2] The run command begins with the word RUN followed by the name of a rule-applying subr and the specification of the

---

[2] The names were chosen purely arbitrarily.

```
DEFINE RUNRULE(RULESET,ARPASPELL)
    DO DONESET:=EMPTY;
       TRYSET:=SET OF (ARPASPELL);
     X:IF EMPTY(TRYSET) THEN GO D;
       SPELL:=CHOICE OF(TRYSET);
       ADD SPELL TO DONESET;
       REMOVE SPELL FROM TRYSET;
       FOR I:=1 STEP 1 UNTIL I>LENGTH(SPELL)
        DO FOR RULE IN RULESET
            DO CHANGES:=RULE(SPELL,I):
               IF CHANGES
                  THEN DO NEWSPELL:=MAKECHANGE(ARPASPELL,
                                                         CHANGES);
                          IF NEWSPELL NOT IN TRYSET AND
                             NEWSPELL NOT IN DONESET
                             THEN ADD NEWSPELL TO TRYSET;
                       END;
            END FOR;
        END FOR;
       GO X;
     D:FOR SPELL IN DONESET DO PRINT(SPELL);
    END RUNRULE;
```

Figure 3
NONDETERMINISTIC RULE APPLICATION ALGORITHM

input string(s). The joe command is the word JOE followed by the specification of the input string(s). JOE is an unordered rule-applying subr that uses all rules that are currently defined. It is automatically recompiled by an implicit subr command whenever it is necessary. The mary command is the word MARY followed by the specification of the input string(s). MARY is a nondeterministic rule-applying subr that uses all the rules that are currently defined. It is automatically recompiled by an implicit subr command whenever it is necessary. All forms of the run commands are terminated by semicolons. When a subr is operated, all derived spellings are output along with the names of the rules that have transformed the string. The forms of the input string specification are described by example.

RUN XYZ (T EH:2 S T);
The rule-applying subr XYZ is operated against the given ARPAbet spelling with exterior word boundaries appended automatically.

JOE TEST;
The rule-applying subr JOE is operated against each form of

the word TEST that is in the lexicon. If necessary, JOE is recompiled.

MARY TEST:2;
The rule-applying subr MARY is operated against the second form of the word TEST from the lexicon. If necessary, MARY is automatically recompiled.

RUN ABC LEX;
The rule-applying subr ABC is operated against each form of each word in the entire lexicon.

RUN JOE LEX FOO;
The rule-applying subr JOE is run against all forms in the sub-lexicon FOO. This command is exactly equivalent to

JOE LEX FOO;

8. OUTPUT COMMANDS

Three commands are available for the output of defined objects and the output of the results of some run commands. Output may be to the user's terminal, printer, or disk. The <output> commands begin with the name of the device: TERMINAL, PRINTER, or DISK. If DISK is used, then a file name is also given.[3] The output options, separated by commas, follow the device (and file) specification. The possible output options and their meanings are:

SUBRS - output the current definition of all rule applying subrs as subr commands.

RULES - output the current definition of all rules as $ commands.

SLEXS - output the current definition of all sub-lexicons as slex commands.

LEX - output the current definition of each lexicon entry (all forms) as lex commands.

ALL - equivalent to the output option sequence LEX, SLEXS, RULES,SUBRS.

---

[3] The knowledgeable LISP user may instead specify a file descriptor list. A file name alone, say FN, is equivalent to the file descriptor list (FN INFIX A W). In any event, if the selected disk output file already exists, it is erased before the command is executed.

SUBR s - output the current definition of the rule applying subr s as a subr command.

RULE r - cutput the current definition of rule r as a $ command.

SLEX l - output the current definition of sub-lexicon l as an slex command.

RUN s l - (where s is a rule-applying subr name including JOE and MARY, and l is a sub-lexicon name or the word LEX) applies s to all forms specified by l; each original and all derived forms are output.

All outout options except run print in a format that is proper for recompilation. Thus, the command,

DISK XYZ ALL;

would output all current definitions to the file XYZ in a recompilable format. To compile the contents of a disk file, use a dcomp command such as

DCOMP XYZ;

The combination of a disk and dcomp command may be used to obviate the necessity of saving the entire system module to preserve work in progress.

Other examples of output commands are

TERMINAL SUBR FOO, RULE FLAP;
PRINTER RUN FOO SL1;

The first command outputs the definition of the rule-applying subr FOO and the rule FLAP to the user's terminal. The second command outputs the results of operating the rule-applying subr FOO against each form in the sub-lexicon SL1 to the high speed printer.

For some uses, the query command is more economical -- see Section 6.

9. DELETE COMMAND

A delete command may be used to remove from the system a rule-applying subr, a rule, a sub-lexicon, or a lexicon entry. The forms of the command are described by examples.

DEL SUBR XYZ;
The rule applying subr XYZ is removed from the system. All associated symbolic data and code are erased.

DEL RULE ABC;
The rule ABC is removed from the system. All associated symbolic data and code are erased. Rule-applying subrs that reference this rule should be edited and recompiled.

DEL SLEX QRS;
The definition of the sub-lexicon QRS is removed from the system.

DEL TOTAL;
The lexicon entry TOTAL and all of its forms are deleted from the lexicon. Sub-lexicons that reference this word or any of its forms should be edited and recompiled.

10. THE EDITOR AND THE RECOMPILE COMMAND

This section describes a mini-editor that may be used for correcting input and a command for entering edit mode with current definitions.

10.1 THE EDITOR

The editor is automatically entered when a syntax error is detected or when certain other error conditions occur. The input to the LISP INFIX compiler (and hence commands to the rule system) are viewed as token strings. Examples of tokens are identifiers, unsigned numbers, and delimiter characters such as colon, plus (+), and equal (=). During compilation, tokens are input and added to a "last tokens input list". When an error is detected, this list is available for editing. If input is from a terminal, the part of the input line past the point at which the error is detected is lost. If input is from any other device, the current character and token positions are maintained as a point at which further input may be found after editing.

Upon entry to the editor, an optional error message is output along with the last several tokens on the last tokens input list. If no specific error message is issued, the problem is some general syntax malady, such as using a comma for an identifier. During editing, two token lists are maintained -- last (initially the last tokens input list) and next. At the completion of editing, last is appended to the front of next. The combined list is then used as the input source for the compilation. If the command is not completed in this list, more tokens are input from the file that was in use when the error occurred. If a further error occurs, the editor is re-entered. If the error occurs while using the list, then the tokens up to and including the error are on the last tokens input list (last) and the remaining tokens are on next. Parts of both last and next are output upon editor entry if they are not empty.

Several commands are available in the editor to manipulate last and next: ML, MN, PL, PN, DL, DN, AL, and AN. The commands T and E are also available to continue or abort the compilation. Multiple commands may appear on one line, and a single command may stretch over multiple lines. The following paragraphs describe the commands.

10.1.1 ML and MN Commands

ML is followed by a positive integer. The specified number of tokens are moved from next to last. MN is used in a like manner to transfer tokens from last to next. Given the following initial values of last and next:

```
last = SUBR FOO ALLOF )
next = A , B ) , C ;
```

the command ML 2 would produce

```
last = SUBR FOO ALLOF ) A ,
next = B ) , C ;
```

and the command MN 2 would produce

```
last = SUBR FOO
next = ALLOF ) A , B ) , C ;
```

10.1.2 PL and PN Commands

PL and PN are followed by a positive integer. The specified number of tokens on last or next, respectively, are printed. Given the initial configuration

```
last = $ FLAP DX = VOWEL / T
next = OR D / VOWEL ;
```

the command PL 3 would output

```
VOWEL / T
```

and the command PN 3 would output

```
OR D /
```

10.1.3 DL and DN Commands

DL and DN are followed by a positive integer. They delete the specified number of tokens from last and next, respectively. Given the initial values of last and next:

```
last = SLEX SL1 TOTAL
next = , , COULD ;
```

the command DL 2 would produce

```
last = SLEX
next = , , COULD ;
```

and the command DN 2 would produce

```
last = SLEX SL1 TOTAL
next = COULD ;
```

10.1.4 AL and AN Commands

AL and AN are followed by an input sequence. They add the input sequence to last and next, respectively. The input sequence is delimited (on both ends) by any token that does not appear in the sequence. Given the initial values of last and next

```
last = SLEX S2 TOTAL ,
next = COULD , ANY ;
```

the command AL / PRODUCT, / would produce

```
last = SLEX S2 TOTAL , PRODUCT ,
next = COULD , ANY ;
```

and the command AN / PRODUCT, / would produce

```
last = SLEX S2 TOTAL ,
next = PRODUCT , COULD , ANY ;
```

10.1.5 T Command

The T command signals that editing should be terminated and that compiling should commence. The compiler restarts with the tokens in last and next and then returns to the input file for any additional program text. If input is from the user's terminal, additional text may be input on the same line as the T command (or on following lines). If input is from a device other than the terminal, then reading resumes, after exhaustion of last and next, just beyond the point at which the error was detected. For example, suppose that the following line were input from the terminal:

SUBR FOO ONEF) A,B) ,C;

The editor would respond with the error message "MISSING (". The value of last would be SUBR FOO ONEOF), and next would be empty. The remainder of the input line (because it came from a terminal) would be discarded. The remedy would be the sequence of commands:

DL 1 T (A,B) ,C;

DL 1 would delete the erroneous ")", and the T command would initiate the re-compilation. The total token sequence input to the compiler would then be

SUBR FOO ONEOF (A , B) , C ;

10.1.6 E Command

An E command exits (aborts) compilation of the current input. This command is recognized only by the editor. If input is from the terminal, the the command supervisor will be left in a position to accept the next command. If input is from some other device, the rest of the input file is skipped.

10.1.7 General Comments About Editing

ML, MN, PL, PN, DL, and DN commands receive a token count (an integer) as an argument. If the count exceeds the number of tokens in the list (last or next) specified by the command, then the whole list is moved, printed, or deleted, as is appropriate. An input to the system may be broken with a % preceded by a space. If you are in the editor, you will stay there. If not, you will be put into the editor with the message, "ESCAPE".

10.2 THE RECOMPILE COMMAND

A recompile command is used to edit and recompile the definition of a rule-applying subr, a rule, or a sub-lexicon. Assume that S is a subr name, R is a rule name, and SL is a sub-lexicon name. Then three forms of the command are:

```
RECOMP SUBR S;
RECOMP RULE R;
RECOMP SLEX SL;
```

When definitions are input to the system, the symbolic token string is kept "two deep". That is, the latest and next to latest definitions are maintained. The above commands work on the latest definitions. To work on the older definitions, use commands of the form:

```
RECOMP OLD SUBR S;
RECOMP OLD RULE R;
RECOMP OLD SLEX SL;
```

The recomp command causes the specified definition, as a token string, to become the value of the editor's list, next. Last is emptied, and the editor is entered. You may then make any appropriate changes and give a T command to recompile the definition with the modification(s). For example, suppose the following definition is made and used:

```
SUBR FOO ALLOF(A,B),ONEOF(C,D);
```

The command

```
RECOMP SUBR FOO;
```

is given. Then the following edit commands are issued:

```
ML 2 DL 1 AL / ONEOF / T
```

The new definition of FOO is

```
SUBR FOO ONEOF(A,B), ONEOF(C,D);
```

and the original definition is the one useable by the word OLD. To restore the old definition and make the current definition the old one, use the command

```
RECOMP OLD SUBR FOO;
```

Simply give the editor the T command, and the recompilation and swap will occur.

When definitions are output (by an output command), only the current (or latest) definition is printed. The delete command deletes both the current and the old definitions. When definitions are made for which no current definitions exist, the present input becomes both the current and the old definition. If an uncorrected error occurs in a definition (giving the editor the E command) no changes or additions occur in the saved copies.

11. SYSTEM ASPECTS

This section describes some miscellaneous features and capabilities of the phonological rule testing system as opposed to the command language. Even though many of the discussed items are of interest only to one using the system as a function library, others using the system can benefit from a quick reading of this section.

The phonological rule testing system is embedded in SDC LISP. The commands are implemented as an extension of the LISP INFIX language. It is possible to intersperse rule system commands with input to the LISP compiler and evaluator. Therefore, some commands that contain syntax errors may be interpreted as LISP. When this happens, the resulting error messages and/or evaluations are based upon the standard LISP rules.

The following subsections describe interaction with the operating system, compiler switches, functions that process ARPAbet, and the subr execution support routines.

11.1 INTERACTION WITH THE OPERATING SYSTEM

SDC LISP and hence the phonological rule testing system operate under the VM/370 monitor using CMS -- see [3]. Terminal connection is made to the system via either a telephone or the ARPA network -- see [4]. The following describes the procedures using a telephone connection. Section 11.1.7 describes the differences using the network.

11.1.1 Logging in and Loading

After you dial VM on a telephone line, the system outputs the herald "VM/370 ONLINE" and enters an idle state. To login, hit the carriage return and wait for the output of "!". Then type a login command.

    L user pass

User is the user's account name and pass is a password associated with that account. (The login and all other commands and input lines are terminated by a carriage return.) The system responds to the login with the output of a variety of greeting and informative messages. You are now logged into the system and may issue commands to the monitor.

At this point, you will wish to load and use the rule-testing system. This is possible only if the RULELIB

191 disk is attached to your login. If it is not, then enter the command sequence,

```
LINK RULELIB 191 198 RR
ACC 198 B
```

To load the system, input the command

```
TESTRULE
```

The system is loaded, and LISP outputs its set of greeting messages. A state has been reached where you may now enter rule language commands and LISP expressions.

11.1.2 Line Editing Characters

The operating system provides a minimal line editing capability. The input of certain characters affects the composition of the line. The default line editing characters and their meanings are: @ (delete this and the previous character from the input line); [(delete this character and all previous characters from the input line); # (logical end of line -- used to input two or more logical lines on the same physical line); and " (accept the next character as is -- do not interpret it as a line editing character).

As luck would have it, each of the chosen four line editing characters has a usage in the rule testing language or INFIX LISP. Therefore, it is strongly recommended that some such command as

```
TERM CH { LINED _ LINEN OFF ES OFF
```

be entered to the monitor (not to LISP -- perhaps before you load the rule-testing system). The result of the above TERM command is to make "{" the character delete instead of "@", "_" the line delete instead of "[", and to turn the logical line end and escape character facilities off.

11.1.3 Prompts and Breaks

VM/370 handles all terminal traffic (network or telephone) in half-duplex mode. Therefore, input should not be entered unless it is expected by the monitor. When input is expected, a prompt character is output. The default prompt is a bell. If desired, the prompt may be changed by giving the monitor (not LISP) the command

```
TERM READ c
```

where c is a non-numeric character. After this command, c will be output instead of a bell whenever input is expected.

If there is input when not expected, or the break key (attention button on 2741 terminals) is depressed at any time, then the monitor is entered. AT this point, "!" is output and one of four actions may be taken: (1) input a carriage return (program execution will resume); (2) enter HT (terminal output will cease until the next terminal input request is issued); (3) enter RT (cancels the last HT command and resumes terminal output); and (4) enter HX (execution of the currently loaded program is permanently discontinued after output of any stacked terminal lines). If instead of one break, two are input in reasonably rapid succession, then CP is output and you are in a position to interact with the CP monitor component. To resume execution of the loaded program, input "BEGIN".

11.1.4 Monitor Commands from LISP

Terminal lines input to LISP that begin with "/" have a special interpretation. If the first two characters are "//", then the CMS subset mode is entered. Any CP or CMS subset commands may be input and executed. To return control to LISP, input the command "RETURN".

If the first character of an input terminal line is "/" and the second character is not, the the entire line except for the first character is passed through to CP for execution. For instance, to send a message to the terminal of the user with account name JOHN, input

```
/M JOHN message text
```

to LISP. If you are talking directly to the monitor, the command is entered without the "/". Note, the above usage of "/" to communicate with the monitor can conflict with the delimitation of the nucleus portion of a rule definition. For the latter use, make sure it is not the first character input on a terminal line -- if necessary, type a blank first.

11.1.5 LISP Return and Logging Out

To return from LISP (and the rule testing system), enter the command

```
RCMS;
```

This returns you to the state that you were in before loading the system. The monitor command to log out of VM/370 entirely is

LOG

Before returning to the system, it may be desired to save the current work. This can most easily be accomplished using the output commands -- see section 8. Another method is available that saves the entire module, currently in operation, on disk. Assume that it is desired to save the module as MYTESTER;* then input the command

SAVE("MYTESTER);

The module is written to disk and may later be reloaded by the monitor command

MYTESTER

Any identifier of eight or fewer characters that does not contain any special characters may be used as the module save name instead of MYTESTER. The save command is an example of an ordinary LISP expression that is not part of the rule testing language. The quote mark used in the save command informs LISP that the name is an identifier datum and not a variable name.

A module save takes more than 300,000 bytes of disk. Therefore, it should not be used promiscuously.

11.1.6 Errors and Warnings

Besides syntax errors, other anomalies may be detected by the compiler and run time support package. If a rule-applying subr definition that contains references to undefined or deleted rules is compiled, a warning diagnostic is issued -- the subr is still compiled. If a subr containing such a reference is executed, an error message is output and an error state is entered.

Entry to an error state (usually following an execution time infraction) is announced by the output of a message that characterizes the offense and the question

PRINT UNWRAP Y/N/D?

---

* GENMOD is used and will save the file as (MYTESTER MODULE A1).

As a naive LISP user, just enter N or NO and control will return to command input.[5]

11.1.7 Network Usage

The IBM 370 Model 145 at System Development Corporation is connected to the ARPA network as Host 8 and is known as SDC-LAB. Connection from a Terminal Interface Processor (TIP) necessitates usage of the two commands

```
@T O L
@I 8
```

If the connection is succesfully opened, the herald "VM/370 ONLINE" is output along with a prompt. (The default prompt character for network users is a dot.) A login command should now be given. To send a break through a TIP, use the command

```
@S B
```

The break is not acted upon until any stacked terminal lines have been printed -- therfore, be patient. As may be seen from the above TIP commands, "@" has special significance. In order to send "@" through, it is necessary to type "@@". An alternate methcd is to define another character as "@" to VM. For instance, the command

```
SET INPUT } 7C
```

causes the input of "}" to be translated to the EBCDIC character code 7C, which is the internal representation of "@". Output is not affected.

The normal logout command "LOG" automatically closes the network connection. If you drop the connection by any other method, the job is put in a disconnected state and after a respectable length of time is forced off the machine. It is urged that whenever possible, the "LOG" command be used so as to not tie up resources.

For network connection procedures from other than a TIP, consult the TELNET documentation for your local host.

---

[5] A Y or YES response outputs a stack backtrace before returning to command state. A D response enters a special debug supervisor that evaluates LISP S-expressions. To exit debug state, use the word EXIT.

11.2 COMPILER FLAGS

Three flags (LISP variables) are available to control to some extent the amount of editing text saved and the amount of compiler output. The flag RFLG:65 determines whether symbolic definitions of rules are saved for use by later recompile and output commands. Initially, the flag is on. To turn it off, enter

```
RFLG:65=NIL;
```

The flag SFLG:65 determines whether symbolic definitions of rule-applying subrs are saved for later recompile and output commands. Initially, the flag is on. To turn it off, enter

```
SFLG:65=NIL;
```

The flags may be turned back on by the commands

```
RFLG:65=T;
SFLG:65=T;
```

The flag TFLG:65 determines whether the compiler prints results of the compilation of rules and rule-applying subrs. Initially, the flag is off. To turn it on, enter

```
TFLG:65=T;
```

To turn it off again, enter

```
TFLG:65=NIL;
```

If the flag is on, then the original input and the transformations produced by each pass of the compiler are output.

11.3 ARPABET SPELLING PACKAGE

Internally, ARPAbet spellings are maintained as integer arrays with each phone represented by a four byte (32-bit) integer. The information in each byte is

byte 0 - kind, voicing, and stress level
byte 1 - consonant class
byte 2 - consonant place of articulation
byte 3 - 8-bit representation of phone's name

The redundancy in the representation is used for a speed advantage by the rules.

A spelling is built using the functions CHECKSPELL and

MAKESPELL. CHECKSPELL has one argument, a list of phone names, ":", and integers. If the list is legal -- e.g., all list items are phone names, ":" only follows full vowel names, integers only follow ":", and all integers are 0, 1, or 2 -- then CHECKSPELL returns the list in a "normal" form with exterior word boundaries appended. If the list is not legal, error messages are output, and NIL is returned. The function MAKESPELL takes as an argument a normal form value of CHECKSPELL and converts the representation to an integer array.

The spellings of forms associated with a word may be retrieved using the macro SPELL. For example,

```
SPELL("TOTAL)
```

returns a list of the forms of the word TOTAL. Each form is an integer array. Given an integer array spelling, the (identifier) name of the Ith phone is retrieved by

```
GETNAME(s,I)
```

where s is the array. Assuming that the Ith phone is a vowel, its stress level (an integer) is retrieved by

```
GETSTRESS(s,I)
```

The macros NCODE, PCODE, and FCODE are available to examine the structure of an integer phone representation. The argument to NCODE is a phone name -- the value is the phone's 8-bit name code. The argument to PCODE is a phone name -- the value is the 32-bit integer representation of that phone (sans stress level). The argument to FCODE is a feature name (STRESS0, STRESS1, and STRESS2 for stress levels, and STRESS for the entire STRESS field) -- the value is a list of two integers: (1) the byte in which information for that feature is maintained (0, 1, 2, or 3), and (2) a bit-mask that may be used for extracting all relevent feature information from the selected byte.

The array C2PHARY has 256 elements, each element corresponding to an 8-bit phone code. If p is the name of a phone, then the expression

```
C2PHARY[PCODE("p)+1] EQ "p
```

is always true.

The function PRNSPL has one argument, an integer spelling array. It prints the spelling in a compressed form; i.e., no blanks or ":" are output. It also prints (on the same line if possible) the elements in the list RULES. The value of RULES usually is the set of rules that have performed transformations to produce this spelling. Another function, PRNMAP, behaves similarly to PRNSPL. PRNMAP has no arguments. The array printed is the value of the variable SPELLARY and the number of phones output is equal to the value of the variable SPELLEN.

Caution should be exercised when using the above functions and macros. They do little error checking and can, if misused, lead to unrecoverable errors. All are name protected through the section mechanism. The names CHECKSPELL, MAKESPELL, SPELL, GETNAME, GETSTRESS, NCODE, PCODE, and FCODE are in section 1; C2PHARY is in section 65; and PRNSPL and PRNMAP are in section 66. Thus, for example, it is necessary to enter NCODE:1 rather than just NCODE. See the next section for full names of the variables SPELLARY, SPELLEN, and RULES.

11.4 EXECUTION SUPPORT PACKAGE

This section briefly describes the necessary protocol to use rules and rule-applying subrs other than through rule-testing commands. The original intended usage of the system was in this mode as a dynamic component of a speech understanding system -- see [2] and [5].

The LISP sectioning mechanism has been used to minimize name conflicts of system components and to aid program organization. The sections used by the phonological rules system are:

- 1 - general utility functions
- 65 - rule, subr, and command compiler internal
- 66 - execution support package
- 67 - ordered rule subrs
- 68 - unordered and nondeterministic rule subrs
- 69 - rule functions
- 113 - compiler command handlers

Section 1 is used by other components of the speech understanding system, and section 113 is used by most language extention facilities in LISP. Unless specifically stated to the contrary, all support functions and variables discussed below are in section 66.

11.4.1 Internal Array Handling

When rules are applied, they may match some part of the input string (an integer array) and transform it. For unordered and nondeterministic rule application, it is imperative that the original string not be damaged. Therefore, new arrays must be allocated to hold the transformed spellings. Because this happens frequently and can cause time consuming garbage collects, an "erasure" scheme has been adopted. To allocate an array, use

```
CREATEARY()
```

The value is an array into which spellings may be copied. To return the array A to a pool for later use, use

```
ERASEARY(A)
```

The pool of available arrays is maintained on the list ERASEL. The length of arrays allocated by CREATEARY is equal to the value of the variable ARRAYLEN. The initial value of ARRAYLEN is 50. To change this value to x, execute

```
[ERASEL=NIL,ARRAYLEN=x];
```

This will ensure that the pool of arrays of the old length is discarded. Note that the size of these arrays should be a little longer than the longest spelling you will ever derive.[6]

Spelling arrays that hold lexicon forms are of the exact length of the spelling (in phones, including initial and final word boundaries). The function COPA may be used to copy a spelling array of an exact length to an array allocated by CREATEARY. If the value of the variable CFLG is NIL, then a copy is not made and the argument is returned. Otherwise, an array is created and the argument is copied into it. The initial value of CFLG is T.

Because the system normally operates with arrays that are longer than the actual spellings, it is necessary to communicate the actual lengths. As rules and rule-applying subrs are operated, they attempt to make the value of the variable SPELLEN the actual number of phones and the value of the variable SPELLARY the array containing the spelling. The variable SPELLINX normally is the index of the phone that starts the currently visible input substring, and the variable NEXTARY is an array in which a rule may perform the reconstruction dictated by its left side.

----

[6] Be generous -- the penalty for exceeding this bound may be an unrecoverable program check.

11.4.2 Rule Calling Sequence

A rule is compiled as a function with the rule name in section 69 as the function name. When a rule is called, it is expected that the phonetic input string be in the array that is the value of the variable SPELLARY and that the value of the variable SPELLINX be the index of the phone that starts the currently visible substring. If the rule does not match the substring, then NIL is returned as the value. If the rule matches, then several conditions prevail when the rule function returns: (1) the value of the function (in register AC) is the length (a small integer) of the reconstruction sequence generated by the rule's left side, (2) the identifier name of the rule is in register AC0, (3) the system entries BMRK and EMRK are set to the absolute locations of the beginning and just beyond the end of the part of SPELLARY matched by the rule's nucleus, and (4) the system entry CHANGES contains the reconstruction sequence generated by the rule's left side. If the rule function returns non-NIL, then the derived spelling is generated by calling the proper reconstruction function. The reconstructor used depends upon whether the rule application is ordered, unordered, or nondeterministic. For ordered rule application, use a code sequence like

```
(ARGS)(CALL rule)(BZM AC (LABEL L))(ARGS)(CALL RECO)L
```

for unordered rule application,

```
(ARGS)(CALL rule)(BZM AC (LABEL L))(ARGS)(CALL RECU)L
```

and for nondeterministic rule application,

```
(ARGS)(CALL rule)(BZM AC (LABEL L))(ARGS)(CALL RECN)L
```

where rule is the rule name in section 69 -- for instance (FLAP . 69). The reconstruction functions RECO, RECU, and RECN each behave a little differently to be compatible with the different kinds of rule applying subrs. Each is described below.

11.4.3 Ordered Subrs

An ordered rule-applying subr is compiled as a two-argument function. The name of the function is the subr name in section 67. The arguments are the input spelling and the length of the spelling. The input spelling is copied by COPA (if the value of CFLG is non NIL). The value of the variable SPELLARY is set to the input spelling, and the value of the variable SPELLEN is set to the spelling's length. Then the subr parts are executed, left to right across the input, one at a time. If a rule in a subr part matches the input substring, then RECO is called immediately. RECO performs several functions: (1) sets the value of the variable RULENAME to the name of the rule matched, (2) makes the changes in SPELLARY, (3) sets SPELLEN to the new spelling length (4) adds the name of the rule (value of RULENAME) to the list RULES, and (5) calls the value of the functional variable MAPPER. In rule testing mode, the value of MAPPER is the function PRNMAP. The variable SPELLINX is the index of the substring matched by the rule. Its value is available when the value of MAPPER is call...

Several points should be noted when any program other than the rule testing system is directly calling an ordered rule-applying subr: (1) it is the caller's duty to bind or set CFLG to the proper value, (2) RULES should be re-bound so that it will reflect only the rules that have operated on this spelling, and (3) if desired and appropriate, call ERASEARY with the value of SPELLARY at the completion of execution.

11.4.4 Unordered Subrs

An unordered rule-applying subr is compiled as a three-argument function. The function name is the subr name in section 68. The arguments are the spelling array (input string), the number one, and the spelling's length. (The second argument is actually used as the substring start location. However, unordered subrs are called recursively and must be initially "primed" with a one.) Unordered subrs re-bind numerous global variables: SPELLARY, SPELLINX, SPELLEN, NEXTARY, RULES, SPELLFN, and RULENAME. When the subr is entered, it immediately calls the value of the functional variable MAPPER (set to PRNMAP in rule testing mode). The values of SPELLARY, SPELLINX, SPELLEN, and RULES are proper and reflect facts about the spelling in SPELLARY. The functional variable SPELLFN is bound to the subr itself so that RECU can make proper recursive calls. The value of NEXTARY is initialized to an array (by CREATEARY) on each subr entry, and is released (by ERASEARY) on subr exit. The array is used by RECU.

RECU is the reconstruction function used by unordered rule-applying subrs. The name of the rule that just matched is stuffed into the variable RULENAME. Then the new spelling is constructed in NEXTARY without modifying the value of SPELLARY. Next, the value of SPELLFN (the subr) is called recursively with the three arguments NEXTARY, SPELLINX, and the length of the new spelling that is in NEXTARY.

When an unordered rule is called, it is not necessary to copy a spelling of exact length into a longer array -- the original spelling array is not altered. Before the subr is called, the value of RULENAME should be bound to some meaningful value, say the word whose spelling is being operated upon. The reason is that the value of RULENAME is added to RULES by the initial subr call as if it were a rule name.

11.4.5 Nondeterministic Subrs

A nondeterministic rule-applying subr is compiled as a three-argument function. The function name is the subr name in section 68. The first argument is the input string (as an integer array), and the third argument is the length of the input string. The second argument is irrelevant. (This makes calling sequences reasonably compatible with unordered rule-applying subr functions.) The subr binds several special variables: SPELLARY, SPELLINX, SPELLEN, NEXTARY, TRYSET, and DONESET. The value of SPELLINX cannot be relied upon in nondeterministic subrs. The value of NEXTARY is initialized to an array by CREATEARY. All arrays created in a nondeterministic subr are erased (by ERASEARY) before exit. Each time a rule matches the input substring, RECN is called. The functions performed by RECN are: (1) stuff the value of RULENAME with the name of the rule that matched, (2) put the derived spelling in NEXTARY, (3) add to TRYSET[7] a list of the length of the derived spelling, NEXTARY, and the name of the rules that have participated in deriving this spelling, and (4) set the value of NEXTARY to another array (using CREATEARY).

Eventually, each array on TRYSET (including the original input array) becomes the value of SPELLARY, the length of the array is set in SPELLEN, and RULES is set to the list of rules that have participated in deriving this spelling. The value of the functional variable MAPPER is then called. In rule testing mode, the value of MAPPER is the function PRNMAP. As with unordered subrs, the value of RULENAME should be bound to some meaningful value before the subr is called. The initial value of RULENAME is added to RULES as if it were a rule name.

---

[7] Step three is bypassed if the same spelling has already been generated.

BIBLIOGRAPHY

[1] - "INFIX LISP FOR SDC IBM 370 USERS", J.A. Barnett, TM-4310/600/00, 5/24/73.

[2] - "A PHONOLOGICAL RULES COMPILER", J.A. Barnett, Proceedings of the IEEE Symposium on Speech Recognition, 4/74, pp. 188-192.

[3] - "IBM VIRTUAL MACHINE FACILITY/370: COMMAND LANGUAGE GUIDE FOR GENERAL USERS", Order Number GC20-1804-n.

[4] - "CURRENT NETWORK PROTOCOLS", Includes BBN Report 1822.

[5] - "A VOICE-CONTROLLED DATA MANAGEMENT SYSTEM", H.B. Ritea, Proceedings of the IEEE Symposium on Speech Recognition, 4/74, pp. 28-31.

APPENDIX 1: COMMAND SYNTAX

The following summarizes the syntax of the commands in the rule system. The description is standard BNF with the usual augmentation; namely, the use of % means that the following term must occur at least once and may occur multiple times. A form like %'x' means that the following term must occur at least once or may occur multiple times separated by x's. For example, %'-'A means A, A-A, A-A-A, etc. Square brackets, [and], mean that the occurrence of the enclosed term is optional. { and } are used as meta parentheses. The occurrence of | between terms means alteration i.e., a choice of the terms.

The syntax of a rule definition is

```
<rule>::=$ <r-name> <left-side>=<right-side>[<conditional>];

<r-name>::=<identifier>

<left-side>::=NIL|%',<left-part>

<left-part>::=<consonant-name>|<boundary-name>|
              <reduced-name>|<left-vowel>|
              <index>|<constructed-consonant>

<left-vowel>::=<vowel-designator>[<stress-designator>]

<vowel-designator>::=<full-vowel-name>|<index>

<index>::=<integer>

<stress-designator>::=<explicit-stress>|<borrow>

<explicit-stress>::= :<stress>

<stress>::=0|1|2|

<borrow>::=@<index>

<constructed-consonant>::=(<class-designator>
                          <place-designator>
                          <voice-designator>)

<class-designator>::=<class-name>|CLASS<borrow>

<place-designator>::=<place-name>|PLACE<borrow>
```

```
<voice-designator>::=[-]VOICE[<borrow>]

<right-side>::=<nucleus>|
               [<left-context>]/[<nucleus>]/
               [<right-context>]

<nucleus>::=%','<right-part>

<right-context>::=%','<right-part>

<left-context>::=%','<right-part>

<right-part>::=<repeat>|<optional>|<choice>

<repeat>::=REP <min-count> <choice>

<min-count>::=<integer>

<optional>::=OPT <choice>

<choice>::=%'OR'<pat-part>

<pat-part>::=<consonant-name>|<boundary-name>|<reduced-name>|
             <full-vowel-name>[<explicit-stress>]|
             <class-name>|<place-name>|<kind-name>|
             VOICE|VOWEL<explicit-stress>|<feature-bundle>|
             +<pat-part>|-<pat-part>

<feature-bundle>::=(%<choice>)

<conditional>::=IF <cond-body>

<cond-body>::=%'OR' <cond-and>

<cond-and>::=%'AND' <relation>

<relation>::=<kind-test>|<class-test>|<place-test>|
             <stress-test>|<name-test>|<voice-test>|
             (<cond-body>)

<kind-test>::=KIND<borrow> {EQ|NQ}
              {KIND<borrow>|<kind-name>}

<class-test>::=CLASS<borrow> {EQ|NQ}
               {CLASS<borrow>|<class-name>}

<place-test>::=PLACE<borrow> {EQ|NQ}
               {PLACE<borrow>|<place-name>}

<stress-test>::=STRESS<borrow> {EQ|NQ|GQ|LQ|GR|LS}
                {STRESS<borrow>|<stress>}
```

```
<name-test>::=NAME<borrow> {EQ|NQ}
              {NAME<borrow>|<boundary-name>|<consonant-name>|
               <reduced-name>|<full-vowel-name>}

<voice-test>::=[-]VOICE<borrow>|
               VOICE<borrow> {EQ|NQ} VOICE<borrow>

<consonant-name>::=L|W|Y|R|NX|N|M|G|B|D|P|T|K|ZH|Z|DH|V|
                   SH|S|TH|F|JH|CH|Q|DX|WH|HH

<boundary-name>::=*|#

<reduced-name>::=AX

<full-vowel-name>::=IY|IH|EY|EH|AE|AA|AH|AO|OW|UH|UW|
                    ER|AW|AY|OY

<class-name>::=AFRIC|FRIC|PLOS|NASAL|GLIDE|
               LATERAL|CENTRAL

<place-name>::=LABIAL|ALVEOLAR|ALVPAL|DENTAL|
               VELAR|PALATAL

<kind-name>::=BOUNDARY|CONST|VOWEL
```

The syntax of a subr definition is

```
<subr>::={<unordered-subr>|<nondeterministic-subr>|
          <ordered-subr>};

<unordered-subr>::=SUBR UNORDERED <s-name>
                            %','<r-name>

<nondeterministic-subr>::=SUBR NODETERM <s-name>
                                 %','<r-name>

<ordered-subr>::=SUBR [ORDERED] <s-name> %','<subr-part>

<s-name>::=<identifier>

<subr-part>::=<r-name>|<allof>|<oneof>|<if>|<unless>|
              (<subr-part>)

<allof>::=ALLOF(%','<subr-part>)

<oneof>::=ONEOF(%','<subr-part>)

<if>::=IF <subr-part> THEN <subr-part> [ELSE <subr-part>]

<unless>::=UNLESS <subr-part> THEN <subr-part>
```

```
                [ELSE <subr-part>]
```

The syntax of a recompile command is

```
<recomp>::=RECOMP [OLD] {SUBR <s-name>|
                         RULE <r-name>|
                         SLEX <l-name>};
```

The syntax of a delete command is

```
<delete>::=DEL {<word>|
                SUBR <s-name>|
                RULE <r-name>|
                SLEX <l-name>};
```

The syntax of a sub lexicon definition command is

```
<sub-lexicon>::=SLEX <l-name> %','{<word>[<form-index>]};

<word>::=<identifier>

<form-index>::= :<integer>
```

The syntax of a lexicon entry definition command is

```
<lexicon>::=LEX {<lex-print>|<lex-aug>|<lex-def>};

<lex-print>::=<word>[<form-index>]

<lex-aug>::=<word>+%','<arpa-spell>

<lex-def>::=<word><form-index><arpa-spell>|
            <word>%','<arpa-spell>

<arpa-spell>::=(%{<consonant-name>|<boundary-name>|
                 <reduced-name>|
                 <full-vowel-name>[<explicit-stress>]})
```

The syntax of the execute commands is

```
<execute>::={<run>|<joe>|<mary>};

<run>::=RUN <s-name> <run-object>

<joe>::=JOE <run-object>

<mary>::=MARY <run-object>

<run-object>::=LEX [<l-name>]|<arpa-spell>|
               <word>[<form-index>]
```

The syntax of the output commands is

```
<output>::={<terminal>|<printer>|<disk>};

<terminal>::=TERMINAL <out-sequence>

<printer>::=PRINTER <out-sequence>

<disk>::=DISK {<f-name>|<fdl>} <out-sequence>

<f-name>::=<identified>

<out-sequence>::=%','<out-spec>

<out-spec>::=ALL|SUBRS|RULES|LEX|SLEXS|SUBR <s-name>|
             RULE <r-name>|SLEX <l-name>|
             RUN {<s-name>|JOE|MARY} {<l-name>|LEX}
```

The syntax of the query command is

```
<query>::=? {RULES|SUBRS|LEX|SLEXS};
```

APPENDIX 2: PHONOLOGICAL SYMBOLS AND THEIR FEATURES

PHONE	EXAMPLE	FEATURES
IY	beat	VOWEL, stress, VOICE
IH	bit	VOWEL, stress, VOICE
EY	bait	VOWEL, stress, VOICE
EH	bet	VOWEL, stress, VOICE
AE	bat	VOWEL, stress, VOICE
AA	bob	VOWEL, stress, VOICE
AH	but	VOWEL, stress, VOICE
AO	bought	VOWEL, stress, VOICE
OW	boat	VOWEL, stress, VOICE
UH	book	VOWEL, stress, VOICE
UW	boot	VOWEL, stress, VOICE
AX	about	VOWEL, :0, VOICE
ER	bird	VOWEL, stress, VOICE
AW	down	VOWEL, stress, VOICE
AY	buy	VOWEL, stress, VOICE
OY	boy	VOWEL, stress, VOICE
Y	you	CONST, GLIDE, PALATAL, VOICE
W	wit	CONST, GLIDE, LABIAL, VOICE
R	rent	CONST, CENTRAL, ALVEOLAR, VOICE
L	let	CONST, LATERAL, ALVEOLAR, VOICE
M	met	CONST, NASAL, LABIAL, VOICE
N	net	CONST, NASAL, ALVEOLAR, VOICE
NX	sing	CONST, NASAL, VELAR, VOICE
P	pet	CONST, PLOS, LABIAL
T	ten	CONST, PLOS, ALVEOLAR
K	kit	CONST, PLOS, VELAR
B	bet	CONST, PLOS, LABIAL, VOICE
D	debt	CONST, PLOS, ALVEOLAR, VOICE
G	get	CONST, PLOS, VELAR, VOICE
HH	hat	CONST, MISC
F	fat	CONST, FRIC, LABIAL
TH	thing	CONST, FRIC, DENTAL
S	sat	CONST, FRIC, ALVEOLAR
SH	shut	CONST, FRIC, ALVPAL
V	vat	CONST, FRIC, LABIAL, VOICE
DH	that	CONST, FRIC, DENTAL, VOICE
Z	zoo	CONST, FRIC, ALVEOLAR, VOICE
ZH	azure	CONST, FRIC, ALVPAL, VOICE
CH	church	CONST, AFRIC, ALVPAL
JH	judge	CONST, AFRIC, ALVPAL, VOICE
WH	which	CONST, MISC, LABIAL
DX	batter	CONST, MISC, ALVEOLAR, VOICE
Q	glottal stop	CONST, MISC, VOICE
*	syllable	BOUNDARY
#	word	BOUNDARY

stress = :0, :1, or :2